

Certified Equivalence for Minecraft Redstone: A Verification Funnel and an Audit of Its Soundness

Mike Murunov and Maksim Borisov

GenProof

{mike, max}@genproof.com

Abstract—A checked certificate can certify the wrong statement. We study this on Minecraft redstone, where tools compile Verilog into circuits but none proves that a build computes its specification. Our gate accepts a layout only with a machine-checkable certificate. SAT proves the graph the engine extracts, under our executable semantics, equivalent to the specification; an LRAT proof is checked by our own RUP checker and by the verified checker `cake_lpr`; Lean 4 re-proves the claim from the same extracted graph. Auditing the gate ourselves found 7 wrong acceptances, several with a checked certificate for the wrong statement, two ways to forge a certificate (one an unsound LRAT checker in `drat-trim`). The model’s link to the game is tested, not proved: in steady state it agrees with two engine backends on 16,873 random layouts and with a vanilla Minecraft oracle on 44 benchmark and 200 random layouts. Generic tests miss weak-carry faults that the proof (and ABC) refutes. At default limits the gate certifies 65 of 99 layouts that three third-party compilers produce from our specifications, and finds wrong layouts in two of them.

Index Terms—Minecraft redstone, equivalence checking, proof certificates, LRAT, Lean, differential testing, AI agents

I. INTRODUCTION

A checked certificate can certify the wrong statement. In the first version of our gate, a plain wire was accepted as an implementation of the constant 0, with a proof that `drat-trim` verified. The proof was correct. A specification input named `g_y` had merged with a net of the gate’s own miter, so the proof was about a different circuit (section V). Most holes we found later were of this kind: not in the solver or the checker, but in the code that decides *what* is proved.

We study this on Minecraft redstone. Players have built adders, CPUs and whole computers out of it, and several tools compile hardware descriptions into it: `MinecraftHDL` [1], `rtl2mc` [2], `REDA` [3], `Redstone-Compiler` [4]. The community catalogues dozens of designs for each logic gate [5]. We wanted an optimiser for such circuits, human or AI, whose output is judged by something it cannot fool. For redstone we found no search in the style of `FunSearch`, `AlphaEvolve` or `ShinkaEvolve` [6], [7], [8] whose output is checked by property-based testing or formal verification.

The compilers above validate their output empirically, in simulation or on a vanilla server. We want a gate that accepts a *simplified* circuit only if it provably still computes the specification. This is the proof-carrying-code idea [9]: the producer is untrusted, the check is cheap and trusted. It works only if the gate is sound, and passing tests is not enough. Figure 1 shows two 17-input OR buses. Both pass 4096

```
L:i0 .. L:i8 . L:i9 .. L:i16 . .
w .. w r1E w .. w w O:y seed
w .. w w w .. w w O:y mutant
```

Fig. 1. Two 17-input OR buses in RSL layout syntax (section III); “.” elides repeated columns. The seed is accepted. The mutant, with the repeater `r1E` replaced by wire, passes 4096 random vectors on the engine; the SAT proof returns the counterexample $i_3=1$, all others 0, and the engine confirms it.

random engine tests; only one is correct. In the other, the signal of the four leftmost levers dies in the wire before it reaches the lamp. The bug shows only when the other 13 inputs are off. A random vector hits that case with probability about 1/8700, so 4096 random vectors miss it 63 % of the time. Structured tests catch this bug, and the funnel’s co-simulation now runs them. Section VII gives a bug that they miss.

This paper makes four contributions.

- 1) A **verification gate** for redstone layouts and a precise statement of what it guarantees (section III). It extracts the graph with the MCHPRS [10] frontend, gives it a small executable semantics, proves equivalence with a Verilog specification by SAT, certifies the proof, and co-simulates on two engine backends. The guarantee covers the steady state of the extracted model; its link to the game is tested, not proved. As far as we know, it is the first tool that gives a checkable equivalence certificate for a redstone build.
- 2) An **audit of the gate** (sections V, VI and VIII): two rounds of adversarial inputs, differential testing against both MCHPRS backends, and a vanilla Minecraft oracle. Every hole comes with its fix and a regression case. We distil them into a threat model and a table of failure classes that we expect to carry over to other pipelines in which a producer proposes and a checker certifies.
- 3) A second **route in Lean**, independent of the Verilog translation and the CNF encoder but not of the extractor (section IV). The gate proves an `Implements` theorem for every layout it accepts. Kernel-only theorems cover four circuit families of every width. An exact tick model gives settle bounds and three sequential circuits.
- 4) An **experimental evaluation** (section VII): a benchmark of gates, adders up to 15 bits, comparators, a multiplier, ISCAS-85 c17 [11] and wide buses; mutation analysis and testing baselines that are fair to testing; layouts from three third-party compilers; and AI agents and a model-

driven search as producers, which serve as motivation and a demonstration, not as a study of search.

Figure 2 shows some of the circuits in the game. The main lesson is not specific to Minecraft. A certificate is only as good as the claim it certifies. Most of the holes were not in the solver or the checker. They were in the glue around them: port lists, names, stale files, and the question of which layouts can be built at all.

II. BACKGROUND

Redstone: A redstone signal has a strength from 0 to 15. Levers, torches and repeaters emit 15. Each block of wire (“dust”) lowers the strength by one, so a signal dies after 15 blocks. A torch is an inverter: it turns off when the block it hangs on is powered. Merging wires is a free OR, and a torch on a block fed by several wires is a NOR. A repeater restores the strength to 15 and delays by 1–4 redstone ticks. A comparator reads the strength behind it (b) and the maximum at its two sides (s). In compare mode it outputs b if $b \geq s$, otherwise 0. In subtract mode it outputs $\max(b - s, 0)$. So a subtract comparator with a strong side input computes $b \wedge \neg s$ in one block; the compact designs of section VII rely on this. Wire, repeaters, comparators, levers and floor torches must stand on a full block, and a wall torch must hang on one.

MCHPRS and redpiller: MCHPRS [10] is a Rust server for computational redstone. Its `redstone` crate ports the vanilla component logic; we call it the *vanilla-port* backend. Its compiler, *redpiller*, turns a world into a graph. Nodes are components; an edge carries the wire distance and a type, *default* or *side*. A backend interprets this graph much faster than the block-level engine. We pin MCHPRS at commit `a056d7f3` and use it as a library.

Certified equivalence: Two combinational circuits are equivalent if their *miter* can never output 1. The miter feeds both circuits the same inputs and ORs the XORs of their outputs. Checkers reduce this to SAT [12]. An UNSAT answer is only as trustworthy as the solver. Solvers such as CaDiCaL [13] therefore emit a clausal proof (DRAT) that a small checker, `drat-trim` [14], replays. The LRAT variant adds hints that make checking simple enough for verified checkers [15], [16].

III. THE VERIFICATION FUNNEL

A candidate is a text layout in RSL, a format we designed to be easy for people and AI agents to write (fig. 1). Rows run north to south, tokens west to east, and `layer` starts the next level up. Tokens name blocks: `w` wire, `#` sandstone, `tE` a wall torch lighting east, `r1E` a repeater with delay 1 facing east, `sE` a subtract comparator, `L:a` a lever that is input a , `O:y` a lamp that is output y . A sandstone floor is added under the first layer. The funnel (`tools/verify.py`) has eight stages (fig. 3). It ACCEPTS a layout only if all of them pass.

1 Extract: The `rsv` helper (Rust, 916 lines) builds the layout in an MCHPRS test world as a player would: wire shapes follow the vanilla placement rules. Since the audit it also rejects layouts that cannot be built (section V).

It settles the world with all levers off. It then runs the *redpiller* frontend passes (`IdentifyNodes`, `InputSearch`, `ClampWeights`) with no optimisation. The result is the component graph.

2 Subset and model: `graph2v.py` (156 lines) is the formal semantics. It rejects components outside the supported subset, such as locking repeaters and comparators that read containers. It also rejects cyclic graphs, which are sequential. Every other node becomes a 4-bit Verilog wire, and an edge with distance d delivers $\max(v - d, 0)$:

lever	$15 \cdot in$
torch	15 if $\max(inputs) = 0$, else 0
repeater	15 if $\max(base) > 0$, else 0
comparator	$b = \max(base)$, $s = \max(side)$; compare: $b \geq s ? b : 0$; subtract: $\max(b - s, 0)$
lamp	lit iff $\max(inputs) > 0$

Wire nodes are observers only: *redpiller* already links every source to its consumers with the wire distance. The graph is acyclic, so the steady state is unique. Equivalence means: for every lever assignment, after settling, every lamp equals the specification output of the same name. Strength must be 4-bit, not Boolean. One of our first mutants has a comparator side input one block too long, with strength 14 against 15. It is equivalent under a Boolean abstraction and wrong on the engines.

3 Ports: The specification’s interface is read from Yosys’ elaboration, not parsed from text. Four rules apply. Every port is a 1-bit input or output. The specification has no flip-flops or latches. No port name collides with a name the funnel generates. The ports equal the layout’s lever and lamp names.

4 Prove: A generated miter instantiates the specification and the model. Its internal names carry a reserved prefix. Yosys [17] runs `prep`, then `setundef -undriven -anyseq`. This turns every `x` or undriven bit of the specification into an unconstrained input. The proof must hold for every value it takes. `sat -prove` then either proves the miter output constantly 0 or returns a counterexample. The funnel replays the counterexample on the engine and reports both values.

5 Certify: Yosys writes the same miter as AIGER [18] *without any optimisation pass*. The script is: `read, proc -noot, flatten`, a 20-line rule that makes division by zero unconstrained, `setundef, techmap, aigmap`, and `check -assert` against logic loops. A 110-line trusted encoder, `aig2cnf.py`, turns the AIG into CNF by plain Tseitin encoding. It emits three clauses per AND gate, simplifies nothing and rejects anything non-canonical. CaDiCaL (untrusted) writes an LRAT proof. The proof must pass our own RUP checker and the verified checker `cake_lpr` [16]; without it the gate cannot accept. `drat-trim`’s `lrat-check` runs as a third opinion only, because we found it unsound (section V). 10 files are bound: layout, specification, graph, model, miter, division rule, Yosys script, AIG, CNF and proof. Input hashes are pinned before the solver runs and checked afterwards; the proof is then hashed. `check_cert.py` re-derives the whole chain from the layout byte for byte and re-checks the

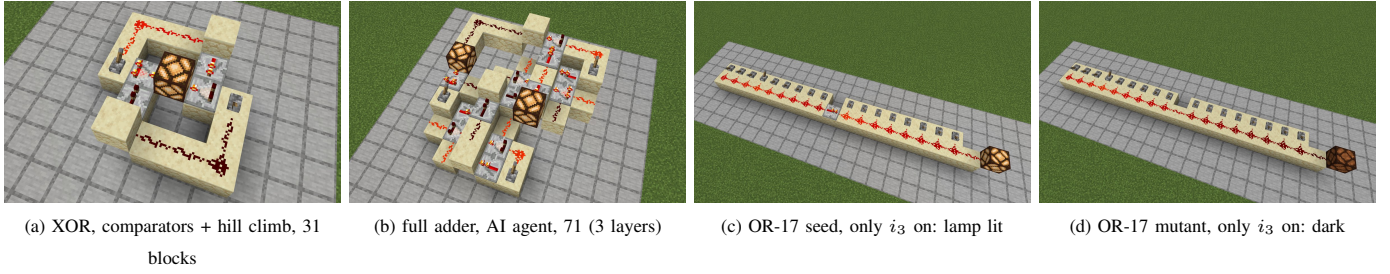


Fig. 2. Benchmark layouts in vanilla Minecraft Java 1.21.1, placed and driven by the client mod (section VI). Inputs are set to the vector that lights the most lamps, except in (c, d), which show the proof’s counterexample for the mutant of fig. 1: 4096 random tests missed it, the game confirms it.

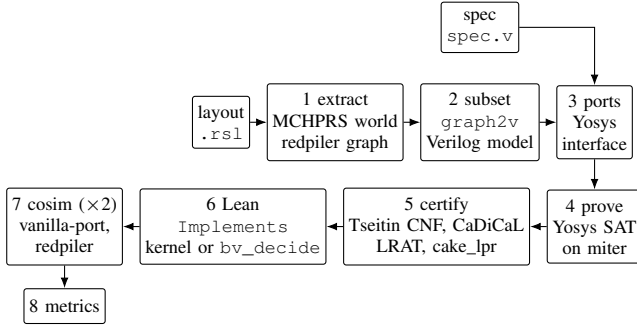


Fig. 3. The funnel. A layout is accepted only if every stage passes.

proof. None of the 44 certificates is trivial. The smallest AIG (not) has 84 AND gates. The largest CNF (add15) has 54,911 variables and 154,379 clauses, with a 4.3 MB LRAT proof.

What an ACCEPT certifies: Suppose the certificate checks. Then, for every lever assignment of the layout with the recorded hash and every value of every undefined bit of the specification with the recorded hash, each lamp of the graph2v model of the layout’s redpiller graph is lit exactly when the specification output of the same name is 1.

The claim has two layers. The *proof layer* says the AIG’s output is 0 on all inputs. It rests only on `aig2cnf.py` and one correct LRAT checker. A bundle of CNF, proof and hashes lets anyone check it with `cake_lpr` alone. The *derivation layer* says that this AIG is the miter of that specification and that model. It rests on the redpiller frontend and `rsv`, on `graph2v.py`, on the miter text written by `verify.py`, and on Yosys’ front end and mapping passes. Section IV bypasses the model’s Yosys mapping, miter text and CNF step; specification mapping remains trusted. Whether the model matches the game is not part of the claim. Co-simulation and the oracles of section VI address it.

6 Lean: The gate proves the same claim again in Lean (section IV). `v2lean.py` (about 200 lines) reads the specification with the certify stage’s Yosys script and prints a Lean specification. Undefined bits become extra quantified inputs, as in the certificate. Lean must prove `Implements` for the layout’s graph against it. Layouts with at most 8 inputs and no undefined bits take a kernel-only route; the others use `bv_decide`. Lean has default step limits and a 30-minute budget; exhausting either is a REJECT and a resource failure,

not a refutation. In make check, 62 accepted layouts and 116 mutants are also run through this stage on their own. 38 of the layouts take the kernel route, in at most 147.9 s each. 24 take `bv_decide`, in at most 21.8 s (add15). All the mutants fail it. It costs 53 % of the suite’s verification time.

7 Co-simulate: The layout runs on the vanilla-port engine and on redpiller. It gets all 2^n lever assignments for $n \leq 16$. Above that it gets $2n+2$ structured vectors (all-zero, all-one, walking one, walking zero), then 4096 fixed pseudo-random ones. For each vector the run must settle within 200 ticks. The lamps must equal the model, which checks the semantics against the engine. They must also equal the specification. With a matching model the proof implies this, so it fails only if the funnel itself is broken. We added it after the audit as a second line of defence.

8 Metrics: Block count (including levers, lamps and the floor), bounding-box volume, component count, critical path in redstone ticks from the model, and settle ticks and glitches from the engine run.

IV. PROOFS IN LEAN

We restated the semantics in Lean 4 (core library only) and proved every accepted layout again there. This path shares nothing with the funnel’s SAT proof except the extracted graph and, in the gate, the reading of the specification.

Semantics and the steady-state theorem:

`Semantics.lean` defines node kinds, links with distance and side flag, 4-bit strengths (`BitVec 4`) and the node functions of section III. A state s is *steady* under inputs x if every node’s value equals its node function applied to what its links deliver. Take any graph with a topological order. We prove once that a steady state exists, is unique, and equals topological evaluation (`steady_exists_unique`, `steady_iff_topoEval`). The funnel only argued this informally. The theorems use no axioms beyond `propext` and `Quot.sound`, and no `sorry`.

Per-layout theorems: `graph2lean.py` writes a layout’s graph as Lean data, independently of `graph2v.py`. It states

$$\begin{aligned}
 \text{Implements } L \ S &::= \text{ports match} \wedge \text{well-formed} \\
 &\wedge \text{topologically sorted} \\
 &\wedge \forall x \ s. \text{Steady } x \ s \rightarrow \text{read } s = S(x).
 \end{aligned}$$

We prove the last conjunct by unfolding it to one 4-bit equation per node and calling `bv_decide` [19]. It bit-blasts, runs CaDiCaL and checks the LRAT proof with a verified checker. For layouts with at most 8 inputs we also prove it by `decide +kernel`, which trusts only the Lean kernel. The specifications are written in Lean by hand and cross-checked against the Verilog. Up to 16 inputs the check is exhaustive. Above that (the wide buses, `add8`–`add15`, `eq12`, `eq16`) it matches the Verilog expression structurally and samples about ten thousand vectors. So the frozen hash does not fully carry over to this route. The gate’s stage 6 closes this gap: its specification comes from `v2lean.py`, and for 26 specifications we prove it equal to the hand-written one. 15 equalities are checked on every input in the kernel, 5 by `simp` and `ac_rfl`, and 6 by `bv_decide`. Three deliberately wrong specifications fail this check. All 49 accepted layouts we translated are proved (33 of them also kernel-only), in 0.6–190.3 s each including Lean start-up; the largest graph (`sevenseg2`) has 464 nodes. All 77 mutants fail with a counterexample that the engine confirms. We also prove, for each of the 116 mutants, that it does *not* implement its specification. On one input vector, the lamps that the Lean evaluator settles to differ from the specification’s output, and the steady state is unique. All 116 refutations go through `decide +kernel`. So a rejection no longer rests on a proof search that happened to fail.

Every width at once: Four generators build the buses, adders and comparators. We wrote each again in Lean and proved `Implements` for every width n , in the kernel only. The bus proofs go by induction over repeater segments, since each repeater restores the signal to 15. The adder proof rests on a tile lemma, checked on all 8 inputs, then induction over tiles and core’s `BitVec` carry lemmas. The comparator proof uses an XOR tile lemma and the bus induction. A script compares the Lean generators with the extracted graphs node for node: 91 of 91 match, the 11 committed layouts plus widths 1 to 20 of all four families. Agreement of the Python and Lean generators is tested only at those widths. With these theorems, only the hand-drawn OR-17 layouts, among the 14 accepted benchmark layouts above 8 inputs, still need `bv_decide`.

Timing: an exact tick model: The steady-state semantics says nothing about delays. A second Lean model (`TickExact.lean`) follows `redpiller`’s scheduler at the pinned commit. Each node has an output, a pending flag and the strength each incoming link delivers. The queue has four time slots and four priorities, each a FIFO. `Redpiller` visits a node’s consumers in an order that comes from a randomly seeded hash map. So this order is an input of the model, and `rsv trace -log` records the order a run used. Our rule is that no semantics enters without co-simulation, so we compared first. On 10,282 recorded runs in 6 campaigns, the model matches `redpiller` on every tick and node of every run. It matches the vanilla-port engine on 10,013 runs, exactly those where the two engines agree with each other. Of the other 269, 197 come from vanilla’s block-update order. With that order, learned per layout and direction from vanilla’s logs, the same model reproduces 10,210 runs. In the remaining 72, a lamp

turns off one or two ticks early after an update that changes none of its inputs; every other node agrees. For 20 layouts, Lean proves three facts under every `redpiller` order and under vanilla’s order. The circuit settles within T redstone ticks (3–8). $T - 1$ ticks are not enough. The lamps then show the specification. Of their lamps, 14 are glitch-free and 18, in 13 layouts, have a glitch witness, including `c17` and the full adder. These checks use `native_decide`; 18 of the 20 layouts also pass kernel-only. Most of these glitches were invisible to the funnel’s co-simulation, which switches inputs only from the all-off state.

Sequential circuits: The funnel handles combinational layouts only. With the exact model we proved three small sequential ones against state machines: an RS latch of two cross-coupled torches, a T flip-flop (the latch plus a comparator edge detector) and a 2-bit counter of two chained flip-flops. The theorem says: after any input sequence in which each change is followed by T ticks, nothing is scheduled and the lamps equal the machine’s output. The proof is an inductive invariant over all reachable pairs of circuit and machine state. A general lemma shows that link steps whose consumers use different queues commute. It reduces the counter’s 18,432 possible `redpiller` orders to one check. The settle times are 3, 5 and 11 redstone ticks, each tight. The latch and the flip-flop also pass kernel-only; the counter needs `native_decide`. On 900 co-simulation runs, half of them violating the settle time, the model matches both engines on every tick and node. A theorem about a circuit that cannot run would be empty. `Check.lean` shows for the XOR and the full adder that a steady state exists for every input, is unique and reads the specification’s output. For the sequential circuits, the invariant’s check reaches every machine state.

Trusted base: On this path the trusted base is the Lean kernel, the compiled `bv_decide` checker, `graph2lean.py`, the `redpiller` frontend, and the hand-written Lean specifications with their cross-check. Lean 4.33 records the `bv_decide` verdict as an extra axiom; the kernel-only route avoids it. Yosys, ABC, the miter text, the CNF encoder and every external proof checker are gone. The gate’s stage 6 trusts `v2lean.py` and Yosys’ elaboration and mapping of `spec.v` instead of the hand-written specifications; the two are proved equal.

Auditing the proofs: None of the 1,976 theorems in the repository uses `sorry` or an axiom of our own. A generated file pins each theorem’s axiom list with `#guard_msgs` in `#print axioms`, so the build fails when any list changes, and the audit fails when a theorem has no guard. Table I counts the theorems by what they trust.

V. AUDITING THE GATE

The funnel shipped with a green regression suite: every seed and candidate accepted, both mutants rejected. We then attacked it the way a search procedure eventually will, with inputs its authors had not intended. Table II lists every case (table III: which stage rejected it). The “before” column comes from running the original code on the same inputs.

TABLE I
THE LEAN THEOREMS BY WHAT #PRINT AXIOMS SHOWS. THE
KERNEL’S OWN AXIOMS ARE PROPEXT, QUOT.SOUND AND
CLASSICAL.CHOICE.

route	theorems	trusts beyond the kernel
kernel only	1,455	nothing
bv_decide	264	compiled LRAT checker
native_decide	257	compiled Lean code

TABLE II
ADVERSARIAL AUDIT OF THE FUNNEL. “BEFORE” IS MEASURED BY
RUNNING THE OLDER CODE ON THE SAME INPUT (EPOCH 0 UNLESS
MARKED 0.1, THE VERSION AFTER THE FIRST FIXES); “NOW” IS THE
VERDICT AND THE FIRST FAILING STAGE. THE FIVE CASES BEFORE THE
LAST WERE FOUND WHILE REBUILDING THE CERTIFICATE. †A LYING
SOLVER: THE CIRCUIT IS CORRECT, BUT THE CERTIFICATE COULD BE
FORGED. THE CORRECT VERDICT IS REJECT EXCEPT FOR “ARITHMETIC IN
SPEC IS CERTIFIED”.

case	before	now
miter name clash (g_y)	ACCEPT + certificate	REJECT @prove
multi-bit output	ACCEPT + certificate	REJECT @ports
comma-continued port list	ACCEPT + certificate	REJECT @ports
x in spec is not a don’t-care	proof passed; then crash	REJECT @prove
undriven spec output	crash	REJECT @prove
sequential spec (flip-flop)	REJECT, wrong reason	REJECT @ports
sequential spec (latch)	crash	REJECT @ports
port names clash with model wires	crash	REJECT @ports
floating wire	ACCEPT + certificate	REJECT @extract
wire on a lever	ACCEPT + certificate	REJECT @extract
duplicate lever name	REJECT, confusing	REJECT @extract
arithmetic in spec is certified	REJECT (no certificate)	ACCEPT
stale certificate in a reused -keep dir	ACCEPT, stale certificate	REJECT @certify
spec defines a module named like the model	REJECT (held)	REJECT @prove
spec defines a module named like the miter	REJECT (held)	REJECT @prove
implicit net in spec	REJECT (held)	REJECT @prove
forged LRAT proof (lying solver)	forged proof accepted†	REJECT @certify
solver rewrites the CNF it was given	certificate for a swapped CNF†	REJECT @certify
combinational loop in spec	proof vacuous; REJECT by accident	REJECT @certify
x from an out-of-range index	REJECT by accident	REJECT @certify
division by zero in spec	ACCEPT + certificate (0.1)	REJECT @certify
verified checker (cake_lpr)	missing ACCEPT, cake_lpr optional	REJECT @certify

Name hygiene: The first miter declared wires g_y and i_y for an output y . A specification input also named g_y became the same net as the gold instance’s output. That net had two drivers, and the miter no longer compared what it was meant to compare. The funnel accepted a plain wire as an implementation of the constant 0, with a DRAT certificate that drat-trim verified. The certificate was correct; it certified the wrong miter. Yosys did print a warning (*Drivers conflicting with a constant driver*), but nothing read it.

Interface extraction: A regular expression parsed the ports. It took the last word of each comma-separated item. output [1:0] y became a 1-bit y , and Yosys resized

TABLE III
DETECTION SUMMARY, NOT AN ABLATION: THE FIRST STAGE THAT
REJECTED EACH CASE. EXT, SUB, PRT, PRV, CER, LEA, COS: EXTRACT,
SUBSET, PORTS, PROVE, CERTIFY, LEAN, CO-SIMULATION. CRS: CRASH OR
TIMEOUT. OUT: CAUGHT ONLY BY A CHECK OUTSIDE THE GATE. ACC:
ACCEPTED (THE RIGHT ANSWER FOR CONTROLS). NONE: NOTHING
NOTICED (FOR TRANSLATOR FAULTS MOSTLY BECAUSE NO TEST LAYOUT
EXERCISES THE FAULT; NONE WAS A WRONG
ACC). N/R: NOT RECORDED; THE FIVE CARRY-SKIP CONTROLS IN COS
ARE WIDER THAN THE ENGINE’S 32-INPUT LIMIT. TRANSLATOR FAULTS
ARE INJECTED: ^a GRAPH2V AND MITER TEXT, FUNNEL RUN; ^b
GRAPH2LEAN AND V2LEAN, LEAN ONLY; ^c PAIRED EDITS.

class	N	ext	sub	prt	prv	cer	lea	cos	crs	out	acc	none
Audit, must reject	21	3	0	5	6	7	0	0	0	0	0	0
Audit, must accept (control)	1	0	0	0	0	0	0	0	0	0	1	0
Shipped mutants	116	0	0	0	116	0	0	0	0	0	0	0
Transl. faults, funnel run ^a	114	0	24	0	35	0	0	1	10	8	0	36
Transl. faults, Lean only ^b	200	n/r	n/r	n/r	n/r	n/r	132	n/r	n/r	12	0	56
Transl. faults, paired ^c	10	0	0	0	5	0	0	0	0	0	0	5
Wire, timing mutants	748	0	14	0	510	0	0	0	0	0	224	0
Carry-skip faults	69	0	0	0	69	0	0	0	0	0	0	0
Carry-skip controls	12	0	0	0	0	0	0	5	0	0	7	0

the connection to one bit with only a warning in its log. In output y , z , the second port had no direction keyword. It disappeared and was never compared. Both were accepted with certificates. The fix reads the interface from Yosys’ own elaboration. Yosys warnings that signal a malformed miter are now fatal: resized ports, conflicting or missing drivers, implicit declarations. This rule alone would have caught the name clash and the truncated bus, but not the lost port.

Undefined values: Without setundef, Yosys’ optimisation passes may replace $1'bx$ by any constant; here it became 0. A specification $y = a ? 0 : 1'bx$ was proved equal to a lamp that is never lit. Only a failed CNF export and a crash in co-simulation stopped the ACCEPT. Whether x is a don’t-care is a policy choice. We chose the strict reading, an unconstrained value, because a frozen specification should say what it means.

Physical validity: MCHPRS’ engine crate never removes a block that has lost its support; the server’s interaction layer does that. So a wire floating in the air, or standing on a lever, is simulated as if it were attached. The funnel accepted a layout whose only path from input to output ran through such a wire. No player can build it: in the game the wire pops off. rsv now applies the support rules of MCHPRS’ is_valid_position plus the vanilla wall-torch rule. It also rejects duplicate port names. The property-based testing (PBT) campaign (section VI) found the same hole independently.

Stale state: With -keep, generated files lived in a user-chosen directory. When exporting the current miter failed, the certify stage found the previous run’s CNF and DRAT files and verified them. The suite runner had the same bug one level up. When the funnel crashed, it read the last run’s JSON report and printed “7/7 as expected”. Both now delete their artifacts

first, and every step checks its own return code.

Second round: the certificate itself: Rebuilding the certify stage (section III) turned up five more cases.

The proof checker was unsound. drat-trim’s `lrat-check` (commit 2e3b2dc) prints `c VERIFIED` for an LRAT “proof” that is a single empty clause without hints. The proof line `3 0 0` is accepted for the satisfiable CNF $(x_1 \vee x_2) \wedge \neg x_1$. The check that rejects a lemma with neither a conflict nor RAT candidates is commented out in its source (`lrat-check.c`, line 161). This regresses a closed upstream issue. We reported it [20] with a one-line fix [21]. The funnel now relies on its own RUP checker and on `cake_lpr`. `lrat-check` stays only as a third opinion.

The solver could swap its input. The checkers re-read the CNF after the untrusted solver had run. A solver that replaced it with a trivially unsatisfiable one received a valid certificate. Hashes pinned before the solver runs close this.

A fix opened a hole. In the first round we added a `techmap` so that arithmetic specifications get certificates. It lowers `x % 0` to a fixed value. A specification with a division by zero was then accepted with a checked certificate. Verilog defines the result as `x`; a 20-line rule now makes it unconstrained.

Two further specifications, a combinational loop and an out-of-range bit select, passed the Yosys SAT proof vacuously. They had been rejected only because AIGER export happened to fail.

Diagnostics: Several failures were right for the wrong reason. A Yosys error was reported as “counterexample: all inputs 0”; `x` and `z` values crashed Python. No verdict was wrong, but an AI agent in the loop reads these messages as feedback. Three more probes held from the start. All 22 cases are regression tests (`make adversarial`); the fixes changed no verdict on our circuits.

VI. VALIDATING THE SEMANTICS BY DIFFERENTIAL PBT

The proof is about the model, so the model must match the game. `tools/pbt.py` generates random small layouts of 3–7 by 2–6 cells, with one or two layers, 1–4 levers and 1–2 lamps. All component types appear in all orientations. For every layout inside the subset, it compares three simulators on every lever assignment: the model in Icarus Verilog [22], the vanilla-port engine and redpiler. This is differential testing [23], [24], [25]. Failing layouts are shrunk by replacing blocks with air, wire or sandstone while the failure persists [26].

First campaign: The generator did not know about support. A partial run reported three model/engine mismatches and 42 redpiler panics (*Exceeded the maximum number of default inputs 255*). All 45 shrunk examples were unbuildable (wire on wire, on a lever, over air): without the optimising `DedupLinks` pass, the frontend’s wire search creates duplicate input links through stacked wire and exceeds the backend’s 8-bit per-strength input counters. On such layouts the two MCHPRS backends also disagreed with each other. These findings concern MCHPRS’ robustness, not real builds.

Second campaign: With a support-aware generator and the new check in `rsv`, we generated 20,000 layouts (seed 11). 16,873 fell inside the supported subset, 1,800 used unsupported components, 1,172 had feedback loops and 155 oscillated. Of the in-subset layouts, 8,828 had at least one non-constant output. Of those, 7,975 contained a torch, repeater or comparator. Graphs had up to 39 nodes (mean 13.4). The model, the vanilla-port engine and redpiler agreed on every lever assignment of every one of them (0 mismatches).

Scope: This validates the semantics against MCHPRS. MCHPRS models neither quasi-connectivity nor torch burnout, and its wire uses a non-vanilla update order. We do not expect these to change the steady state of an acyclic circuit; the vanilla check below covers only such circuits.

Vanilla Minecraft as a second oracle: A 117-line client mod for Minecraft Java 1.21.1 (Fabric) runs a script inside a singleplayer world. It places every block with the state MCHPRS computed, flips levers through the game’s own `onUse` code, reads lamp states back, and takes the screenshots of fig. 2. Probed vectors are reached by real lever uses in Gray-code order. Vanilla agreed with the specification and with MCHPRS on all 1,108 probed vectors of all 44 accepted layouts. We probed 30 layouts exhaustively. For more than five inputs we probed all-zero, all-one, walking one and zero, and 16 random vectors. The 3 mutants we loaded failed in vanilla on 1–8 probed vectors each, including the proof’s counterexample wherever we probed it, and always exactly as MCHPRS predicted. In the game we also placed 200 random layouts (the first 200 qualifying seeds of the second campaign, rule in the artifact; 1–4 levers, 1–2 lamps, at most 33 nodes). All 2,074 lever assignments, each reached by real lever uses, gave the lamp states of the model and of MCHPRS: 0 differences (a second full pass read the same lamp states), which excludes (95 %) a mismatch rate per layout above 1.5 %. As a positive control we changed one block in each of 100 so that the engine’s lamps change; the game showed 569 of the 570 predicted differences. The one miss is the only disagreement with the model among 1,058 mutant vectors, and it depends on the wait: after the engine-derived wait the lamp stayed dark in 19 of 20 copies of that mutant, after 400 ticks it was lit, as predicted, in 20 of 20. The harness needed the same care as the funnel: `/setblock` leaves a comparator’s stored output stale and schedules no update, which fooled our first all-off probe. We now use each comparator twice (the mode toggles and back) and reach every vector by lever uses.

VII. BENCHMARK

Circuits: The specifications were frozen, with SHA-256 hashes recorded, before any layout was searched. *Gates:* NOT, AND, NAND, XOR, XNOR, IMPLY. *Small blocks:* 2:1 multiplexer, 3-input majority, half and full adder, 2-to-4 decoder. *Netlist:* ISCAS-85 c17. *Arithmetic:* ripple-carry adders with 4, 8, 12 and 15 bits (largest: 31 inputs, 2,624 blocks), equality comparators with 8, 12 and 16 bits, and a 2×2 multiplier. *Wide:* OR and AND buses with 17, 24 and 32 inputs. *Computer parts:* 4- and 8-bit ALUs and analog seven-segment decoders.

Mutants drop a repeater, rotate a component or switch a comparator’s mode.

Gate-level *seeds* use textbook torch logic; generated arithmetic and analog decoders also use comparators. Each seed may have compact *candidates*. Larger sets: the Lean proofs cover 49 layouts and 77 mutants, make check 62 and 116 (table V: evidence by set).

Who designed the layouts: The AND, NOT and OR-17 layouts come from the project’s first version. AI agents (Claude Opus 5.5 sub-agents, one run per task; an assistant working with the first author for the first few) designed the other small layouts against the funnel and the tileable adder cell and generators. Each got the RSL documentation, a page of mechanics hints and a fast truth-table tool, and logged every evaluation. The logs are self-reports: 22 designs were accepted after 56 truth-table evaluations and 4 mechanics probes. Every submitted layout passed, so rejection feedback was never exercised. This is a first look at the loop we want to automate, not an evaluation of it.

Search against the gate: ShinkaEvolve [8] driving Claude Sonnet 5.5 found a 10-block AND (our previous best, table IV, was 16) in 4 and 7 generations, against the epoch-0 and the current gate. Proof, LRAT certificate, Lean and the vanilla game accept it. It puts the lever on the torch’s block, the gaming move of table VIII. None of the 34 layouts that the epoch-0 gate accepted fails the current gate.

Results: All 44 of 44 seed and candidate layouts were accepted. Proof plus certificate took 0.08–5.25 s per layout and the whole funnel 0.8–90.5 s, dominated by co-simulation. Compact candidates reduced the block count for 13 circuits, by 23–78 % (table IV). By content, 10 of the 13 best layouts use moves that table VIII calls objective gaming; without such candidates, 6 circuits keep a smaller certified layout, by 22–78 %. All reductions are relative to our own seeds; by bounding-box volume no best layout is clearly smaller than the smallest Minecraft Wiki design of the same function.

Third-party compilers: The gate also checks other tools’ output. An import format embeds a compiler’s own structure file with an I/O map, and the certificate hashes the compiler’s bytes. These tools often end in a wire, not a lamp, so we added *observer outputs*: a wire or component position whose value is strength > 0. This semantic extension came with a recorded decision and a rerun of the PBT campaign (20,000 layouts, epoch-1 generator) first: 0 mismatches with either engine. We ran REDA, Redstone-Compiler and rtl2mc on our specifications. Of 504 imported layouts, 405 are files that Redstone-Compiler ships (400 of them tiles), and all get a certified ACCEPT. Of the 99 compiled from our specifications, 65 do (REDA 19 of 26, Redstone-Compiler 46 of 68, rtl2mc 0 of 5; 44 of the 46 are candidate layouts of four gates). Four REDA layouts (add12, add15, add8 and eq16) pass the proof and the certificate but fail Lean at its default limits (the adders also miss the 200-tick settle limit). With the limits lifted, add8 and eq16 pass (add8 with a 600-tick settle limit); even then `bv_decide` fails on add12 and times out on add15 (in the artifact). The other 30 REJECTs have located causes: 17

Redstone-Compiler worlds never settle, 5 contain a feedback loop, 3 REDA layouts rest dust on nothing, and all 5 rtl2mc layouts use redstone blocks. REDA’s accepted layouts are 3–124 times larger than ours. Redstone-Compiler’s are about our size: its 59-block full adder (wire outputs, no lamps or floor) is the smallest certified one, against our 71. Two tools ship wrong layouts that their own checks pass. REDA’s simulator calls its layouts correct, but it places wire on wire or on a repeater, and vanilla drops such wire. Its XOR, XNOR and OR-17 then compute the wrong function, as a vanilla run confirms; for the OR-17 all 4,096 random vectors pass. Our source audit suggests that Redstone-Compiler checks candidates with input switches in place, then removes them and exports without rechecking; we did not instrument this path. In its exported AND and NAND an inverter then drives its own input. The engines never settle. Vanilla burns the torch out after eight pulses and the circuit freezes; the NAND is then wrong and the AND matches its table. rtl2mc uses redstone blocks as constant power, outside our subset. We reported both upstream [27], [28], [29].

Mutation analysis: We use standard mutation analysis [30], [31]. We applied single-block substitutions to 17 accepted layouts, including air cells next to the circuit. We sampled up to 300 mutants per layout (150 for the widest), 3,991 in all. 1,268 were rejected before the proof: unbuildable, outside the subset, or with a changed interface. 477 were proved equivalent and 2,246 were refuted by the proof. Where co-simulation is exhaustive it caught every refuted mutant, as it must. On the wide circuits, 164 of 289 refuted mutants passed 4096 random vectors; the structured vectors that co-simulation now adds catch all of them (table VI). Across all mutants the model disagreed with the engine 0 times, and a proved mutant disagreed with the specification on the engine 0 times.

Testing baselines: Table VI compares testing, enumeration and proof on 667 mutants that the proof refutes, measured with the LRAT pipeline of section III. About 4096 uniform random vectors on the engine miss 164 of the 289 bus mutants. They also miss 47 of 60 mutants of the 32-input equality comparator, which outputs 1 only when $a = b$. A structured set of $2n+2$ vectors (all-zero, all-one, walking one, walking zero) kills all of these in about 0.1 s. It misses 2 of 60 mutants of the 31-input adder, which random vectors catch. Neither family suffices alone; together they catch all of these refuted mutants. So above 16 inputs the funnel’s co-simulation now runs both, structured vectors first. This catches all 14 shipped mutants that random vectors alone passed. Exhaustive bit-parallel simulation of the extracted model, the object the proof reasons about, detects every mutant. At 31–32 inputs it needs a median of 0.42–1.0 s to the first counterexample. All 2^{32} vectors of an accepted 32-input bus take 23–24 s on one Linux core; the 6–8 days of fig. 4 are *engine* time. ABC’s `cec` decides each of these miters in under 1 s. The proof and the LRAT certificate take a median of 0.53 s per bus mutant and 2.9–10 s per mutant of the 31- and 32-input circuits. On that sample, proof adds an independently checked certificate, not detection power over ABC. A separate carry-skip adder

TABLE IV

BENCHMARK. BLOCKS COUNT EVERY BLOCK, INCLUDING LEVERS, LAMPS AND FLOOR. “BEST” IS THE FEWEST BLOCKS AMONG THE ACCEPTED CANDIDATES LISTED (HAND-DESIGNED, AGENT, HILL-CLIMB), NOT A CLAIM OF OPTIMALITY. TIME: PROVE + CERTIFY OF THE BEST LAYOUT ON AN IDLE AWS SERVER (AMD EPYC 7R13, 32 vCPU, LOAD AT MOST 1.4). CNF AND PROOF SIZES ARE OF THE BEST LAYOUT. MUTANTS: SHIPPED MUTANTS (A DROPPED REPEATER; FOR AND2 A LONGER SIDE WIRE) REJECTED / TOTAL, AND IN PARENTHESES HOW MANY OF THE REJECTED PASS CO-SIMULATION (EXHAUSTIVE UP TO 16 INPUTS, ELSE STRUCTURED PLUS 4096 RANDOM VECTORS); – MEANS NONE SHIPPED. COMPONENTS, CRITICAL PATH (REDSTONE TICKS, MODEL; CRIT. RT) AND GLITCHES (ENGINE, ALL VECTORS) ARE SEED/BEST: COMPACT LAYOUTS TEND TO BE SLOWER AND GLITCHIER.

circuit	in/out	seed	best	comp.	crit. rt	glitches	prove+cert s	CNF vars/cl.	LRAT kB	mutants (tests pass)
not	1/1	9	4	1/1	1/1	0/0	0.09	86/254	4	–
and2	2/1	25	16	3/2	2/2	0/0	0.10	248/701	11	1/1 (0)
nand2	2/1	21	7	2/2	1/1	0/0	0.10	212/593	9	–
xor2	2/1	51	31	3/4	2/2	0/0	0.13	798/2231	37	–
xnor2	2/1	53	32	4/5	3/3	0/0	0.14	825/2312	39	–
imply2	2/1	23	5	1/1	1/1	0/0	0.10	185/512	7	–
mux2	3/1	43	18	4/5	3/3	1/0	0.12	550/1532	25	–
maj3	3/1	89	29	4/9	2/3	0/0	0.15	1010/2816	48	–
half_adder	2/2	96	34	6/7	2/4	0/1	0.15	988/2753	50	–
full_adder	3/2	236	71	14/13	5/7	2/8	0.21	1932/5342	95	–
dec2to4	2/4	124	27	8/7	2/3	0/1	0.14	934/2651	47	–
c17	5/2	162	41	12/7	3/4	6/8	0.17	1294/3602	60	–
mul2	4/4	504	–	44	13	7	0.74	8845/24950	566	4/4 (0)
add4	9/5	699	–	60	12	769	1.20	14697/41327	812	5/5 (0)
add8	17/9	1399	–	120	20	12858	2.53	29323/82445	1824	5/5 (0)
add12	25/13	2099	–	180	28	19658	4.02	43949/123563	3085	5/5 (0)
add15	31/16	2624	–	225	34	24756	5.25	54911/154379	4275	5/5 (0)
eq8	16/1	481	–	37	7	0	0.76	9535/26600	497	4/4 (0)
eq12	24/1	721	–	55	9	0	1.14	14381/40106	779	4/4 (0)
eq16	32/1	961	–	73	11	0	1.58	19227/53612	1073	4/4 (0)
or17	17/1	73	56	1/0	1/0	0/0	0.36	4091/11237	208	1/1 (0)
or24	24/1	99	–	1	1	0	0.46	5675/15572	253	1/1 (0)
or32	32/1	131	–	2	2	0	0.61	7502/20573	393	2/2 (0)
and24	24/1	245	–	28	5	0	0.52	6548/18167	283	3/3 (0)
and32	32/1	325	–	37	6	0	0.68	8735/24224	383	4/4 (0)

experiment exposes faults missed by the tests. At widths 12–14, 20 of 45 faulty layouts pass the gate’s co-simulation and both generic structured suites. The proof refutes all 45, with counterexamples confirmed on the engine; all 6 fault-free controls receive full ACCEPT. ABC also finds every fault, as do a fault-specific structured suite and 10^6 random model vectors. At widths 15–24, 10^7 random model vectors miss 3 of 24 faults; above 15 bits the engine’s 32-input limit prevents replay and full ACCEPT.

Certified local search: The mutation analysis also turned up *smaller* accepted mutants. A greedy hill climber (`tools/hillclimb.py`) repeatedly moves to the smallest accepted single-block mutant. It is the simplest search that uses the funnel as its only judge. Every intermediate layout carries a proof and a certificate. From the 4 torch-logic seeds it found nothing smaller. From the 13 compact candidates, it improved 6 of them by 1–2 blocks each, after 298–1,071 funnel runs per candidate. Every accepted step replaces a wire by a sandstone block, which needs no floor block. The climber refines structure; it does not invent it.

Scaling: Figure 4 compares the time of the proof plus certificate with exhaustive simulation on the vanilla-port engine for OR and AND buses of growing width. Proof plus certificate grows from 0.14 s to 0.65 s between 4 and 32 inputs, with CNFs of up to 8735 variables. Exhaustive simulation of AND20 (1,048,576 vectors) took 160 s on one backend; extrapolated to 32 inputs, engine enumeration would take 6–8

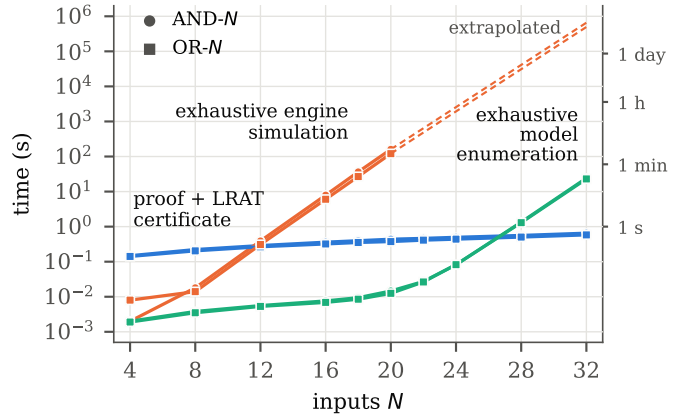


Fig. 4. Proof plus LRAT certificate (median of 3), exhaustive engine simulation, and exhaustive bit-parallel model enumeration (median of 5) for OR- N and AND- N buses. Sequential runs on an otherwise idle AMD EPYC 7R13 host. Dashed engine curves extrapolate beyond 20 inputs.

days.

Hard miters: We checked ripple adders against carry-lookahead and Kogge–Stone specifications, and array multipliers against a Wallace tree, all frozen before evaluation (table VII, 600 s CPU budget). Adders stayed easy. For multipliers, the gate’s Lean route fails first, at 14 inputs on Lean’s heartbeat limit (at 16 on `simp`’s step limit), a resource limit

TABLE V

EVIDENCE BY OBJECT SET: LAYOUTS PASSING THE STAGE/TOTAL, THEN REFUTED MUTANTS/TOTAL. SAT: SOLVER DECIDED; LRAT: CAKE_LPR-VERIFIED UNSAT (LAYOUTS THAT REACH IT); LEAN: PROVED OR REFUTED; COSIM: MCHPRS ENGINES AGREE WITH THE MODEL (*a* EXHAUSTIVE, *s* SAMPLED); VANILLA: REAL MINECRAFT. $A \subset B \subset C$ BY LAYOUT PATH, SO ROWS ARE NOT ADDITIVE. – NOT APPLICABLE; N/R NOT RECORDED.

object set	SAT	LRAT	Lean	cosim	a+s	vanilla	a+s	overlap
A. Benchmark table (Tab. IV): 25 circuits, 44 + 48 = 92	44/44+48/48	44/44	44/44+48/48	44/44+48/48	46a+46s	44/44+3/48	31a+16s	$\subset B \subset C$
B. Per-layout Lean run: 49 + 77 = 126	49/49+77/77	49/49	49/49+77/77	49/49+77/77	72a+54s	48/49+3/77	32a+19s	$\supset A, \subset C$
C. make check suite: 62 + 116 = 178	62/62+116/116	62/62	62/62+116/116	62/62+116/116	100a+78s	48/62+3/116	32a+19s	$\supset A, B$
PBT: 20,000 generated layouts, – 16,873 in the supported subset	–	–	–	16,873/20,000	16,873a+0s	200/200	200a+0s	generated; none in A–C
Third-party, compiled from our specifications: 99 layouts, 65 accepted	69/99	69/99	65/99	66/99	60a+6s	–	–	tool outputs; specs from C
Generator families (or, and, adder, eq) for every width <i>n</i>	–	–	$\forall n: 4/4$ thm., 11/11 lay.	–	–	–	–	11 layouts $\subset A$
Hard miters: 44 (layout, spec) pairs + 105 mutants	41/44+105/105	41/44	24/44+42/105	20/44	n/r	–	–	26 layouts, 14 in C
Exact tick-model runs: 10,282 (5,410 on generated layouts)	–	–	20/41 lay.	10,282/10,282	vp 10,013	–	–	41 layouts $\subset C$

TABLE VI

DETECTION OF MUTANTS THAT THE PROOF REFUTES, BY METHOD. BUS: SINGLE-BLOCK MUTANTS OF THE OR-17, OR-24 AND AND-24 SEEDS; ADD31, EQ32: THE FIRST 60 REFUTED SINGLE-BLOCK MUTANTS OF THE 31-INPUT RIPPLE ADDER AND THE 32-INPUT EQUALITY COMPARATOR; SHIP: THE 48 HAND-MADE MUTANTS. ALL METHODS ALSO DETECT A SAMPLE OF 210 REFUTED MUTANTS OF THE ≤ 5 -INPUT CIRCUITS. ENGINE METHODS RUN ON THE VANILLA-PORT ENGINE. STRUCTURED: ALL-ZERO, ALL-ONE, WALKING ONE AND WALKING ZERO ($2n+2$ VECTORS). CO-SIMULATION: THE STRUCTURED SET, THEN 4096 RANDOM VECTORS. MEDIAN TIME PER MUTANT, SINGLE RUNS ON AN APPLE M4 UNDER LOAD.

method	detected				median s	
	bus	add31	eq32	ship.	bus	31–32
	289	60	60	48		
4096 random (engine)	125	60	13	34	2.0	18–47
0, 1, walking 1/0 (engine)	289	58	60	48	0.08	0.13–0.28
+ 2-hot/2-cold (engine)	289	60	60	48	0.18	3.2–6.7
co-simulation: both sets	289	60	60	48	2.0	17–49
bit-parallel enum. (model)	289	60	60	48	0.01	0.42–1.0
ABC & cec (unverified)	289	60	60	48	0.06	0.07–0.10
SAT proof (Yosys)	289	60	60	48	0.08	0.59–2.0
certify: CNF + LRAT	289	60	60	48	0.44	2.3–8.2

rather than the CPU budget; the other stages pass that layout. Yosys reaches its limit at 18 inputs; ABC and CaDiCaL do at 20 to 22, or never on the array specification. Against its own architecture, the certificate reaches the 32-input limit of `rsv`: 1.4 M variables, a 297 MB proof, 168 s. Refuting a wrong layout stays easy: the SAT proof refutes each of the 105 adder and multiplier mutants in at most 61 s, even at 32 inputs. A separate three-seed campaign extends certificate measurements: at 22 inputs, median CaDiCaL time is 1825 s against `a*b`, versus 10.4 s against the layout’s array architecture. Checker time scales approximately with LRAT size (empirical log-log slopes 0.86–1.00), while proof search and size grow sharply across architectures. Bit-parallel enumeration is faster throughout the measured 10–22-input cross-architecture range, but yields no checked proof. These are certificate-route results, not full-gate ACCEPTS; timings use a shared host.

TABLE VII

HARD MITERS: THE SMALLEST ARRAY MULTIPLIER (IN INPUTS) AT WHICH A METHOD NEEDS MORE THAN 600 S OF CPU TIME ON A 32-CORE SERVER. COLUMNS: SPECIFICATION OF THE SAME (ARRAY) ARCHITECTURE, YOSYS’ READING OF *, AND A WALLACE TREE WITH A KOGGE–STONE ADDER. – = NEVER, UP TO 32 INPUTS (ARRAY) OR 22 INPUTS (OTHERS). ADDERS NEVER EXCEED 44 S.

method	array	*	Wallace
Lean route (<code>bv_decide</code>)	14	14	14
Yosys sat (stage 4)	18	18	18
ABC & cec (unverified)	–	20	20
our RUP checker	–	20	–
CaDiCaL LRAT	–	22	22
cake_lpr (verified)	–	–	–

Independent checks: A separately written Rust encoder agrees on 62 accepted and 116 refuted miters, with all 62 independent UNSAT certificates checked by `cake_lpr`. Translator mutation tests exercise 314 single-site and 10 paired faults, with no observed false ACCEPT: 227 are detected (20 only by differential regressions outside the gate), 68 change no layout of the test pool, and 29 are noticed by nothing. A separate wire/timing generator produces 748 mutants, including 510 proof refutations and 224 steady-state equivalences; co-simulation misses 4 refuted wide-layout mutants.

VIII. LESSONS FOR CERTIFICATE PIPELINES

Threat model: Three parties can make the gate lie. The *producer* is an untrusted search, such as an AI agent or a hill climber. It controls layouts only. The *specification author* is trusted but fallible, and specifications are frozen before search. The *environment* is the gate’s own state: files, tool versions, earlier runs. The first round (table II) found six wrong ACCEPTS. The producer can reach two, the floating wire and the wire on a lever, which share one root cause. Three need a malformed specification: the name clash, the truncated bus and the lost port. One comes from the environment: the stale certificate. The second round adds a wrong ACCEPT on

TABLE VIII
FAILURE CLASSES OF A CERTIFICATE PIPELINE, WHERE THEY SIT, THE
INSTANCE FOUND HERE, AND THE GENERIC CHECK.

class	instance here	generic check
interface binding	truncated bus; lost port in output y, z	take ports from the tool’s elaboration; assert every output is compared
name hygiene	gold output merged with an input net	build the miter structurally; make conflicting-driver warnings fatal
undefined values	1’bx read as a don’t-care	state the policy; encode x as a free input
physical realizability	floating wire accepted	validate the artifact against the platform’s rules before extraction
certificate binding	proof reused from an earlier run	content-address every artifact; check hashes, not file presence
vacuity	optimiser decided the miter; 5-byte proof	plain encoding; treat a very short proof as a warning, not a verdict
diagnostics	Yosys error shown as a counterexample	the gate’s messages are feedback to the search; test them too
oracle hygiene	stale comparator output in the vanilla harness	scrutinise the validation harness like the gate
trusted checker	lrat-check accepts an empty proof	use a verified checker; test checkers with forged proofs
input binding	solver swaps the CNF it was given	pin hashes before untrusted steps run
regression fix	techmap made $x \% 0$ a constant	rerun the whole adversarial suite after every fix
objective gaming	levers on lamps; blocks instead of wire to save floor	the objective is a specification too

the specification side (the division by zero that a first-round fix introduced), two ways for an untrusted solver to forge a certificate (an unsound checker, a swapped input) and two vacuous proofs rejected only by accident.

How the holes were found: An AI assistant (Claude) working with the first author read the gate stage by stage and asked what input would make a stage report success without checking its claim; the 16 first-round probes took about an hour. We make no completeness claim.

The glue is the risk: Table VIII generalises the audit. Apart from one unsound proof checker, no failure was in the solver or the checker; nearly all were in the code that decides *what* is proved, as [32] found for verified distributed systems, and the reason certified encodings exist [33], [34]. Two classes are specific to search: a producer finds whatever the gate does not check, including physically impossible artifacts, and optimises the objective as written, which is how specification gaming arises [35], [36].

IX. RELATED WORK

HDL to redstone: MinecraftHDL [1] synthesises Verilog with Yosys into torch-based standard cells; PERSHING [37] places and routes BLIF netlists; rtl2mc [2], REDA [3] and Redstone-Compiler [4] are recent open-source flows. rtl2mc and REDA check their output against simulation and vanilla servers; none emits an independently checkable equivalence certificate. A 2026 study checks logic gates in a digital twin by truth tables [38].

LLM-driven search with automatic evaluation: FunSearch, AlphaEvolve and ShinkaEvolve [6], [7], [8] evolve

programs scored by automatic evaluators; CircuitEvo [39] and Evolve [40] apply this to logic circuits and Verilog, enforcing correctness by construction or by tests. This gate is a certificate-producing evaluator for such loops.

Certified reasoning: Resolution proofs for combinational equivalence are established [41], [42]. DRAT/LRAT proofs and their checkers [14], [15] are standard in SAT competitions; verified LRAT checkers exist in ACL2, CakeML and Isabelle-LLVM [43], [16], [44]. Translation validation [45], [46] checks each translator output; SAT-based exact synthesis [47], [48] proves gate networks smallest, which we do not. Unlike translation validation of a compiler, the statement being proved here is itself produced by glue code, and that glue is where we found the holes.

X. LIMITATIONS AND FUTURE WORK

Timing and sequential circuits. The gate certifies the steady state only. The exact tick model is tied to the engines by co-simulation and by reading the MCHPRS source, not by proof, and does not model vanilla’s early lamp turn-off. The sequential results cover three small circuits; the counter’s proof needs `native_decide` (kernel evaluation ran out of memory at 55 GB). Outside Lean, witness certificates [49], [50] are the natural route for sequential equivalence.

Oracles. The 20,000-layout campaign compares only with MCHPRS; the vanilla game checks the benchmark and 200 random layouts, none with loops or burnout.

Metric and composition. Counting all blocks rewards levers next to lamps and wire replaced by blocks. The certificate establishes steady-state equivalence of the extracted model, not substitutability under changes to I/O geometry, loading or timing.

Search. The AI-agent and ShinkaEvolve results are few runs by one model with self-reported logs; a controlled study is future work.

XI. CONCLUSION

A redstone layout can carry a proof that it computes its specification, and a verified checker checks that proof in seconds, even for a 15-bit adder. The harder lesson is that a correct certificate can be about the wrong statement: most holes we found were in the code that decides what is proved, and would have let a search collect certified but meaningless wins. Generic tests miss weak-carry faults (ABC still finds them); the proof adds a certificate and independence from any fault model. The game agreed with every certified layout we loaded into it. The implementation, certificates and scripts are in the artifact.

ACKNOWLEDGMENTS

Claude Code 2.1 (Opus 5.5) [51] wrote most code, ran experiments, assisted the audit and drafted text throughout this paper. Codex [52] assisted experiment integration and revision. The first author wrote the first sentence of the introduction. The authors set goals, made decisions, reviewed all text and retain responsibility for it.

REFERENCES

- [1] A. Penhale, O. B. Mashmos, F. O'Brien, and B. H. Meyer, "MinecraftHDL: A digital synthesis flow for Minecraft circuits," McGill University, ECSE 457 design project, Tech. Rep., 2017, <https://github.com/itsfrank/MinecraftHDL>.
- [2] NTP17, "rtl2mc: SystemVerilog to Minecraft redstone," <https://github.com/NTP17/rtl2mc>, 2026.
- [3] Pr0gCat, "REDA: Redstone EDA," <https://github.com/Pr0gCat/REDA>, 2026.
- [4] Redstone-Compiler contributors, "redstone-compiler," <https://github.com/Redstone-Compiler/redstone-compiler>, 2026.
- [5] Minecraft Wiki, "Redstone circuits/logic," https://minecraft.wiki/w/Redstone_circuits/Logic, 2026, accessed 2026-09-28.
- [6] B. Romera-Paredes *et al.*, "Mathematical discoveries from program search with large language models," *Nature*, vol. 625, no. 7995, pp. 468–475, 2024.
- [7] A. Novikov *et al.*, "AlphaEvolve: A coding agent for scientific and algorithmic discovery," arXiv:2506.13131, 2025.
- [8] R. T. Lange, Y. Imajuku, and E. Cetin, "ShinkaEvolve: Towards open-ended and sample-efficient program evolution," in *International Conference on Learning Representations (ICLR)*, 2026, also arXiv:2509.19349. [Online]. Available: https://proceedings.iclr.cc/paper_files/paper/2026/hash/7886b9baf676c52fd568db10ff9772df-Abstract-Conference.html
- [9] G. C. Necula, "Proof-carrying code," in *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1997, pp. 106–119.
- [10] MCHPRS contributors, "MCHPRS: Minecraft high-performance redstone server," <https://github.com/MCHPR/MCHPRS>, 2026, commit a056d7f3, 2026-09-20.
- [11] F. Brglez and H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran," in *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, 1985.
- [12] A. Mishchenko, S. Chatterjee, R. K. Brayton, and N. Eén, "Improvements to combinational equivalence checking," in *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2006, pp. 836–843.
- [13] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froylyks, and F. Pollitt, "CaDiCaL 2.0," in *Computer Aided Verification (CAV 2024)*, ser. LNCS, vol. 14681, 2024, pp. 133–152.
- [14] N. Wetzler, M. J. H. Heule, and W. A. Hunt, Jr., "DRAT-trim: Efficient checking and trimming using expressive clausal proofs," in *Theory and Applications of Satisfiability Testing (SAT 2014)*, ser. LNCS, vol. 8561, 2014, pp. 422–429.
- [15] L. Cruz-Filipe, M. J. H. Heule, W. A. Hunt, Jr., M. Kaufmann, and P. Schneider-Kamp, "Efficient certified RAT verification," in *Automated Deduction – CADE 26*, ser. LNCS, vol. 10395, 2017, pp. 220–236.
- [16] Y. K. Tan, M. J. H. Heule, and M. O. Myreen, "cake_lpr: Verified propagation redundancy checking in CakeML," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2021)*, ser. LNCS, vol. 12652, 2021, pp. 223–241.
- [17] C. Wolf and J. Glaser, "Yosys – a free Verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip 2013)*, Linz, Austria, Oct. 2013, published under the name Clifford Wolf. [Online]. Available: <https://yosyshq.net/yosys/files/yosys-austrochip2013.pdf>
- [18] A. Biere, "The AIGER and-inverter graph (AIG) format version 20071012," Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria, FMV Reports Series, Technical Report 07/1, Oct. 2007. [Online]. Available: <https://fmv.jku.at/papers/Biere-FMV-TR-07-1.pdf>
- [19] H. Böving *et al.*, "Interactive bitvector reasoning using verified bit-blasting," *Proceedings of the ACM on Programming Languages*, vol. 9, no. OOPSLA2, pp. 3259–3285, 2025, the bv_decide tactic (Lean 4 core, Std.Tactic.BVDecide; formerly <https://github.com/leanprover/leasant>).
- [20] M. Murunov, "lrat-check says 'VERIFIED' for a proof that proves nothing (regression #31)," drat-trim issue #42, <https://github.com/marijnheule/drat-trim/issues/42>, 2026, opened 2026-09-29.
- [21] —, "lrat-check: reject an empty lemma that does not end in a conflict," drat-trim pull request #43, <https://github.com/marijnheule/drat-trim/pull/43>, 2026, open on 2026-09-29.
- [22] S. Williams, "Icarus Verilog," <https://steveicarus.github.io/iverilog/>, 2026, version 13.0.
- [23] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.
- [24] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011, pp. 283–294.
- [25] Y. Herklotz and J. Wickerson, "Finding and understanding bugs in FPGA synthesis tools," in *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020, pp. 277–287.
- [26] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of Haskell programs," in *Proc. 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000, pp. 268–279.
- [27] M. Murunov, "build_circuit writes a redstone dust with no block under it (xor2, xnor2, or17)," REDA issue #1, <https://github.com/Pr0gCat/REDA/issues/1>, 2026, opened 2026-09-29.
- [28] —, "build_circuit: refuse a layout with an unsupported component," REDA pull request #2, <https://github.com/Pr0gCat/REDA/pull/2>, 2026, open on 2026-09-29.
- [29] —, "and2 and nand2 output worlds contain a torch loop through an input wire," Redstone-Compiler issue #56, <https://github.com/Redstone-Compiler/redstone-compiler/issues/56>, 2026, opened 2026-09-29.
- [30] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.
- [31] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.
- [32] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy, "An empirical study on the correctness of formally verified distributed systems," in *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. Belgrade, Serbia: ACM, 2017, pp. 328–343.
- [33] S. Gocht, R. Martins, J. Nordström, and A. Oertel, "Certified CNF translations for pseudo-Boolean solving," in *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), K. S. Meel and O. Strichman, Eds., vol. 236. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 16:1–16:25.
- [34] C. Codel, J. Avigad, and M. J. H. Heule, "Verified encodings for SAT solvers," in *Proceedings of the 23rd Conference on Formal Methods in Computer-Aided Design (FMCAD 2023)*, A. Nadel and K. Y. Rozier, Eds. Ames, IA, USA: TU Wien Academic Press, 2023, pp. 141–151.
- [35] V. Krakovna, J. Uesato, V. Mikulik, M. Rahtz, T. Everitt, R. Kumar, Z. Kenton, J. Leike, and S. Legg, "Specification gaming: The flip side of AI ingenuity," DeepMind Blog, Apr. 2020, published 21 April 2020. Accessed 2026-09-28. [Online]. Available: <https://deepmind.google/discover/blog/specification-gaming-the-flip-side-of-ai-ingenuity/>
- [36] J. Skalse, N. H. R. Howe, D. Krashenninnikov, and D. Krueger, "Defining and characterizing reward hacking," in *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds. Curran Associates, Inc., 2022, pp. 9460–9471.
- [37] Q. Nguyen, "PERSHING: An automatic place-and-route tool for Minecraft redstone circuits," in *SIGTBD 2016*, MIT CSAIL, Cambridge, MA, 2016. [Online]. Available: <https://sigtbd.csail.mit.edu/pubs/2016/paper4.pdf>
- [38] D. Cruz García, I. Alonso Correa, S. García González, A. Álvarez Sánchez, and G. Villarrubia González, "Digital twin for designing logic gates in Minecraft through automated circuit verification and real-time simulation," *Electronics*, vol. 15, no. 3, p. 499, 2026.
- [39] Y. Bai *et al.*, "Evolving graph structured programs for circuit generation with large language models," in *International Conference on Learning Representations (ICLR)*, 2026. [Online]. Available: https://proceedings.iclr.cc/paper_files/paper/2026/hash/3b6d138cdcb217ba3c05bd71ee9a33fd-Abstract-Conference.html
- [40] W.-P. Hsin, R.-H. Deng, Y.-T. Hsieh, E.-M. Huang, and S.-H. Hung, "EvoLVE: Evolutionary search for LLM-based Verilog generation and optimization," arXiv:2601.18067, 2026.
- [41] S. Chatterjee, A. Mishchenko, R. K. Brayton, and A. Kuehlmann, "On resolution proofs for combinational equivalence," in *Proceedings of the 44th Design Automation Conference (DAC 2007)*. San Diego, CA, USA: ACM, 2007, pp. 600–605.

- [42] A. Kuehlmann and F. Krohm, “Equivalence checking using cuts and heaps,” in *Proceedings of the 34th Design Automation Conference (DAC ’97)*. Anaheim, CA, USA: ACM, 1997, pp. 263–268.
- [43] M. J. H. Heule, J. Warren A. Hunt, M. Kaufmann, and N. Wetzler, “Efficient, verified checking of propositional proofs,” in *Interactive Theorem Proving – 8th International Conference, ITP 2017*, ser. Lecture Notes in Computer Science, M. Ayala-Rincón and C. A. Muñoz, Eds., vol. 10499. Brasília, Brazil: Springer, 2017, pp. 269–284.
- [44] P. Lammich, “Fast and verified UNSAT certificate checking,” in *Automated Reasoning – 12th International Joint Conference, IJCAR 2024, Part I*, ser. Lecture Notes in Computer Science, C. Benzmüller, M. J. H. Heule, and R. A. Schmidt, Eds., vol. 14739. Nancy, France: Springer, 2024, pp. 439–457.
- [45] A. Pnueli, M. Siegel, and E. Singerman, “Translation validation,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1998)*, ser. LNCS, vol. 1384, 1998, pp. 151–166.
- [46] G. C. Necula, “Translation validation for an optimizing compiler,” in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2000, pp. 83–95.
- [47] A. Kojevnikov, A. S. Kulikov, and G. Yaroslavtsev, “Finding efficient circuits using SAT-solvers,” in *Theory and Applications of Satisfiability Testing (SAT 2009)*, ser. LNCS, vol. 5584, 2009, pp. 32–44.
- [48] W. Haaswijk, M. Soeken, A. Mishchenko, and G. D. Micheli, “SAT-based exact synthesis: Encodings, topology families, and parallelism,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 4, pp. 871–884, 2020.
- [49] E. Yu, A. Biere, and K. Heljanko, “Progress in certifying hardware model checking results,” in *Computer Aided Verification (CAV 2021)*, ser. LNCS, vol. 12760, 2021, pp. 363–386.
- [50] N. Froleyks and E. Yu, “Hardware model checking certification with Certifaiger and Cerbtora,” in *Automated Reasoning (IJCAR 2026)*, ser. LNCS, vol. 16688, 2026, pp. 276–285, tool: <https://github.com/Froleyks/certifaiger>.
- [51] Anthropic, “Claude code,” <https://code.claude.com/docs/en/overview>, 2026, version 2.1, Opus 5.5; use disclosed in Acknowledgments.
- [52] OpenAI, “Codex,” <https://openai.com/codex/>, 2026, used for experiment integration and manuscript revision.